Efficient and Generalized Decentralized Monitoring of Regular Languages

Yliès Falcone, Tom Cornebize, Jean-Claude Fernandez

Efficient and Generalized Decentralized Monitoring of Regular Languages

Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez
Univ. Grenoble Alpes, LIG, VERIMAG, F-38000 Grenoble, France

Abstract. This paper proposes an efficient and generalized decentralized monitoring algorithm allowing to detect satisfaction or violation of any regular specification by local monitors alone in a system without central observation point. Our algorithm does not assume any form of synchronization between system events and communication of monitors, uses state machines as underlying mechanism for efficiency, and tries to keep the number and size of messages exchanged between monitors to a minimum. We provide a full implementation of the algorithm with an open-source benchmark to evaluate its efficiency in terms of number, size of exchanged messages, and delay induced by communication between monitors. Experimental results demonstrate the effectiveness of our algorithm which outperforms the previous most general one along several (new) monitoring metrics.

1 Introduction

Monitoring is a verification technique based on runtime information. From a practical perspective, a decision procedure, the so-called *monitor*, analyzes a sequence of events (or a trace) from the system under scrutiny, and emits verdicts w.r.t. satisfaction or violation of a specification formalized by a property. Being lightweight is an important feature of monitoring frameworks because the performance of the system should be disturbed in a minimal way. When the monitor collects events from a monolithic system, we refer to this as *centralized monitoring*.

Modern systems are in essence distributed: they consist of several computation units (referred to as components in the sequel), possibly interacting together, and evolving independently. Monitoring distributed systems is a long-standing problem. The main challenge is to design algorithms that allow to i) efficiently monitor computation units of a system, ii) let local monitors recompute a global state of the system with minimal communication, and iii) monitor against rich specifications. Existing monitoring frameworks usually assume the existence of a central observation point in the system to which components have to send events to determine verdicts; as seen for instance in [1, 2]. In that case, from a theoretical perspective, monitoring reduces to the centralized case. A more challenging situation occurs when such central observation point cannot be introduced or used in the system. Introducing a central observation point implies to modify the architecture of the system, which is unrealistic in many application domains mainly for economic reasons. Using a central observation point (i.e., one of the components) is also undesirable because it induces i) more communication, ii) unbalanced overhead between components, and iii) more risks of total failure in case of failure of a component. When no such central observation point exists in the system, we refer to this as *decentralized monitoring*. In the decentralized setting, monitors emit verdicts with incomplete information: local monitors read local traces, i.e., incomplete versions of the global trace, and have to communicate with each other to build up a global verdict.

Related Work. Several approaches exist for monitoring distributed systems. A temporal logic, MTTL, for expressing properties of asynchronous multi-threaded systems was presented in [3]. Its monitoring procedure takes as input a *safety* formula and a partially-ordered execution of a parallel asynchronous system. MTTL augments linear temporal logic (LTL) [4] with modalities related to the distributed/multi-threaded nature of the system. Several works like [5] target physically distributed systems and address the monitoring problem of partially-ordered traces, and introduce abstractions to deal with the combinatorial explosion of these traces. Close to our work is an approach to monitoring violations of invariants in distributed systems using knowledge [6]. Model-checking the system allows to pre-calculate the states where a violation can be reported by a process alone. When communication (i.e., more knowledge) is needed between processes, synchronizations are added. Both [6] and our approach try to minimize the communication induced by the distributed nature of the system but [6] i) requires the property to be stable (and considers only invariants) and ii) uses a Petri net model to compute synchronization points. We do not assume any model of the system, i.e., we consider it as a black box. Decentralized monitoring is also related to diagnosis of discrete-event systems which has the objective of detecting the occurrence of a fault after a finite number of steps, see for instance [7, 8]. There are two main differences between monitoring and diagnosis. In diagnosis, a specification with normal and faulty behavior is an input to the problem. Also, when considering observability of distributed systems, diagnosis assumes a central observation point which may not have full access to information. On the contrary, decentralized monitoring does not assume a central observation point, but that local monitors have access to all local information. Similarly, decentralized observation [9] uses a central observation point in a system that collects verdicts from local observers that have limited memory to store local traces. Note, neither diagnosis nor observability considers minimizing the communication overhead.

In [10], we proposed a decentralized monitoring algorithm for (all) LTL formulas. The main novelties were to i) avoid the need for a central observation point in the system and ii) try to reduce the communication induced by monitoring by minimizing the number of messages exchanged between monitors. The approach in [10] uses LTL specifications "off-the-shelf" by allowing the user to abstract away from the system architecture and conceive the system as monolithic. The algorithm relied on a decentralized version of *progression* [11]: at any time, each monitor carries a temporarily extended goal (aka an "obligation") which represents the formula to be satisfied according to the monitor that carries it. The monitor rewrites its obligation according to local observations and goals received from other monitors. According to the propositions referred in the obtained formula, it might communicate its local obligation to other monitors. Our approach relied on the perfect synchrony hypothesis (i.e., neither computation nor communication takes time) where communication relied on a synchronous bus. This hypothesis is reasonable for certain critical embedded systems e.g., in the automotive domain (cf. [10] for more arguments along this line). Moreover, it has been recently shown that this approach does not only "work on paper" but can be implemented when finding a suitable sampling time such that the perfect synchrony hypothesis holds [12].

Nevertheless, to facilitate the application of [10] in more real scenarios, several directions of improvement can be considered. First, it is assumed in [10] i) that at each

time instant, monitors receive an event from the system and can communicate with each others, and ii) that communication does not take time. Second, the approach used LTL formulas to represent the local state of the monitor and progression (i.e., formula rewriting) each time a new event is received. A downside of progression, is the continuous growth of the size of local obligations with the length of trace; thus imposing a heavy overhead after 100 events. Finally, while [10] minimizes communication in terms of number of messages (i.e., obligations), it neglects their (continuously growing) size, with the risk of oversizing the communication device, in practice.

Originality. In this paper, we propose to overcome the aforementioned drawbacks of [10] and make important generalization steps for its applicability. First, instead of input specifications as LTL formulas we consider ("off-the-shelf") finite-state automata and can thus handle all regular languages instead of only counter-free ones. Thanks to the finite-word semantics of automata, we avoid the monitorability issues induced by the infinite-word semantics of LTL [13–15]. Interestingly, algorithms using an automata-based structure are more runtime efficient than those using rewriting (in terms of consumption of time and memory). While our algorithm generally doubles the number of exchanged messages, it reduces the size of messages, the execution time and memory consumption of local monitors by several orders of magnitude. Note, our algorithm is generic: by modifying some of its parameters, one can influence the aforementioned monitoring metrics. Second, in practice, communication and reception of events might not occur at the same rate or the communication device might become unavailable during monitoring. Our algorithm allows desynchronization between the reception of events from the system and communication between monitors but also arbitrarily long periods of absence of communication, provided that a global clock exists in the system. Our algorithm is fully implemented in an open-source benchmark. Our experimental results demonstrate that our algorithm i) leads to a more lightweight implementation, and ii) outperforms the one in [10] along several (new) monitoring metrics.

Overview of the decentralized monitoring algorithm. Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of system components. Let L be a regular language formalizing a requirement over the system global behavior, i.e., L does not take into account the system structure. Let $\tau_i = \tau_i(0) \cdots \tau_i(t)$ be the local behavioral trace on component C_i at time $t \in \mathbb{N}$. Further, let $\tau = \tau_1(0) \cup \ldots \cup \tau_n(0) \cdot \tau_1(1) \cup \ldots \cup \tau_n(1) \cdots \tau_1(t) \cup \ldots \cup \tau_n(t)$ be the global behavioral trace, at time $t \in \mathbb{N}$, obtained by merging local traces. (An hypothesis of our framework is the existence of a global clock in the system.) From L, one can construct a *centralized monitor* for L, i.e., a decision procedure having access to the global trace τ and emitting verdict $\top$ (resp. $\bot$) whenever τ is a good (resp. bad) prefix for L, i.e., whenever $\tau \cdot \Sigma^* \subseteq L$ (resp. $\tau \cdot \Sigma^* \subseteq (\Sigma^* \setminus L)$. Then, from a centralized monitor, we define its *decentralized version*, i.e., a monitor keeping track of possible evaluations of a centralized monitor when dealing with partial information about the global trace. A copy of the decentralized monitor is attached to each component. Our decentralized monitoring algorithm orchestrates message-based communication between monitors. Monitors exchange information about their received events or their evaluation of the current global state. Communication is assumed to be reliable (no message losses) but is not synchronized with the production of events on the system: when a monitor sends a message, there is no special assumption about the arrival time, except that it is finite.

The decentralized monitoring algorithm evaluates the global trace τ by reading each local trace τ_i of C_i, in separation. In particular, it exhibits the following properties.

- If a local monitor yields the verdict $\bot$ (resp. $\top$) on some component C_i by observing τ_i, it implies that $\tau \cdot \Sigma^* \subseteq \Sigma^* \setminus L$ (resp. $\tau \cdot \Sigma^* \subseteq L$) holds. That is, a locally observed violation (resp. satisfaction) is, in fact, a global violation (resp. satisfaction).
- If the monitored global trace τ is such that $\tau \cdot \Sigma^* \subseteq \Sigma^* \setminus L$ (resp. $\tau \cdot \Sigma^* \subseteq L$), at some time t, one of the local monitors on some component C_i yields $\bot$ (resp. $\top$), at some time $t' \geq t$ because of some latency induced by decentralized monitoring, whatever is the global trace between t and t'.

Paper Organization. The rest of this paper is organized as follows. Section 2 introduces some preliminaries and notations. Section 3 proposes a generic (centralized) monitoring framework, compatible with frameworks that synthesize monitors in the form of finite-state machines. Section 4 shows how to decentralize a monitor. In Sec. 5, we present how decentralized monitors communicate with each other to obtain a verdict in a decentralized manner. Section 6 describes the relation between centralized and decentralized monitoring. Section 7 presents our benchmark, DECENTMON2, used to evaluate an implementation of our monitoring algorithm. Section 8 presents some perspectives.

2 Preliminaries and Notations

For $i, j \in \mathbb{N}$, the (underlying set associated to the) interval of integers from i to j is denoted by $[i; j]$. The set of finite sequences over a finite set E is noted E^*.

We consider that the global system consists of a set of components $\{C_1, \ldots, C_n\}$, with $n \in \mathbb{N} \setminus \{0\}$. Each component emits events synchronously and has a local monitor attached to it. An event local to component C_i is built over a set of atomic propositions AP_i, $i \in [1; n]$, i.e., the local set of events is $\Sigma_i = 2^{AP_i}$. The set of all atomic propositions is $AP = \cup_{i \in [1;n]} AP_i$. Atomic propositions are local to components by requiring that $\{AP_i \mid i \in [1; n]\}$ is a partition of AP. (Note, this hypothesis simplifies the presentation of the results in the paper but is not an actual limitation of our framework.) The set of all local events in the system is $\cup_{i \in [1;n]} \Sigma_i$, where Σ_i is visible to the monitor at component C_i, $i \in [1; n]$. The global specification refers to events in $\Sigma = 2^{AP}$ and is given by a regular language $L \subseteq \Sigma^*$. Note that the specification does not take into account the architecture of the system and may refer to events involving atomic propositions from several components (i.e., $\Sigma \neq \cup_{i \in [1;n]} \Sigma_i$ in the decentralized case whereas $\Sigma = \cup_{i \in [1;n]} \Sigma_i$ in the centralized one or when there is only one component). We assume that the (regular) language to be monitored is recognized by a deterministic finite-state automaton $(Q, \Sigma, q_{\text{init}}, \delta, F)$ where Q is the set of states, $q_{\text{init}} \in Q$ the initial state, δ the transition function, and $F \subseteq Q$ the set of accepting states.

Over time, for $i \in [1; n]$, the monitor attached to C_i receives a trace $\tau_i \in (2^{AP_i})^*$, a sequence of local events, representing the behavior of C_i. The global behavior of the system is given by a global trace $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$. The global trace is a sequence of pair-wise union of the local events in components traces, each of which at time t is of length $t + 1$ i.e., $\tau = \tau(0) \cdots \tau(t)$, where for $i < t$, $\tau(i)$ is the (i+1)-th element of τ. The sub-sequence $\tau[i; j]$ is the sequence containing the (i+1)-th to the (j+1)-th elements. The substitution of the element at index t in a sequence τ by e is noted $\tau[t|e]$.

3 Centralized Monitoring of (Propositional) Regular Languages

In this section we propose a general framework for centralized monitoring of regular languages. The framework is compatible with the existing monitoring frameworks that synthesize monitors as finite-state machines for propositional regular languages.

In the centralized case, the monitor is a central observation point. Generally speaking, the purpose of the monitor is to determine whether the observed sequence forms a good or a bad prefix of the language being monitored. For this purpose, the monitor emits verdicts in some truth-domain $\mathbb{B}$ s.t. $\{\bot, \top\} \subset \mathbb{B}$ where $\top$ and $\bot$ are two "definitive values" used respectively when a validation (good prefix) and violation (bad prefix) of the language has been found, respectively.

Definition 1 (Good and bad prefixes [16]). *The sets of good and bad prefixes of a language $L \subseteq \Sigma^*$ are defined as:*

$$\mathrm{good}(L) = \{\tau \in \Sigma^* \mid \tau \cdot \Sigma^* \subseteq L\}, \qquad \mathrm{bad}(L) = \{\tau \in \Sigma^* \mid \tau \cdot \Sigma^* \subseteq (\Sigma^* \setminus L)\}.$$

Using good and bad prefixes, we can define the centralized semantic relation $\models_C$ for traces, using, for instance, the truth-domain $\mathbb{B} \stackrel{\text{def}}{=} \{\bot, ?, \top\}$, where the truth-value $?$ indicates that no verdict has been found yet. Given $\tau \in \Sigma^*$, we say that $\tau \models_C L = \top$ (resp. $\bot$) whenever $\tau \in \mathrm{good}(L)$ (resp. $\mathrm{bad}(L)$) and $\tau \models_C L = ?$ otherwise.

Definition 2 (Centralized Monitor). *A centralized monitor is a tuple $(Q, \Sigma, q_0, \delta, \mathrm{verdict})$ where Q is the set of states, $\Sigma = 2^{AP}$ the alphabet of events, q_0 the initial state, $\delta : Q \times \Sigma \to Q$ the complete transition function, and $\mathrm{verdict} : Q \to \mathbb{B}$ is a function that associates a truth-value to each state.*

A monitor is a Moore automaton, processing events from its alphabet, and emitting a verdict upon receiving each event. Monitor-synthesis algorithms ensure that i) for any $\tau \in \Sigma^*$, $\mathrm{verdict}(\delta(q_0, \tau)) = \top/\bot$ iff $\tau \in \mathrm{good} \,/\, \mathrm{bad}(L)$, where δ is extended to sequences in the natural way; ii) for any $q \in Q$, if $\mathrm{verdict}(q) \in \{\top, \bot\}$ then $\forall \sigma \in \Sigma : \delta(q, \sigma) = q$. A centralized monitor is a decision procedure w.r.t. the centralized semantics relation $\models_C$.

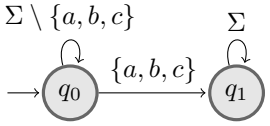

Fig. 1: Transitions of CM_1

Remark 1 (Truth-domains). More involved truth-domains with refined truth-values (e.g., the ones used in [16, 15]) can be used in our framework without any particular difficulty.

Example 1 (Centralized Monitor). Consider $AP^1 = \{a, b, c\}$ and L_1 the language of words over 2^{AP^1} that contain at least one occurrence of the event $\{a, b, c\}$. The monitor CM_1 of this language has its transition function δ_1 depicted in Fig. 1. Moreover, $\mathrm{verdict}(q_0) = ?$ and $\mathrm{verdict}(q_1) = \top$. Consider $\tau_1 = \emptyset \cdot \{a, b\} \cdot \{a, b, c\} \cdot \{a\}$, we have $\emptyset \cdot \{a, b\} \cdot \{a, b, c\} \in \mathrm{good}(L_1)$ and $\tau_1 \in \mathrm{good}(L_1)$.

4 Decentralizing a Monitor

Let us now use the previous example to see what would happen when using a centralized monitor on a local component where only a subset of AP can be observed. Let us consider a simple architecture with three components C_A, C_B, C_C respectively with sets of atomic propositions $AP^1_A = \{a\}, AP^1_B = \{b\}, AP^1_C = \{c\}$. If we use a central

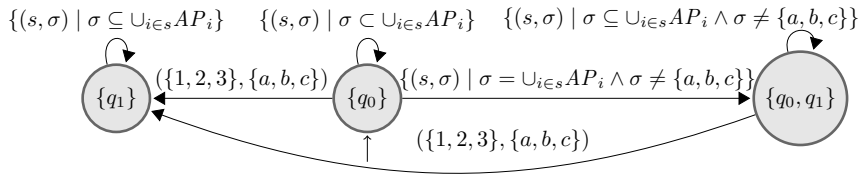

Fig. 2: Transitions of DM_1

monitor on, say C_A, no event (in $2^{AP_A^1}$) could allow the monitor to reach q_1. Monitors should thus take into account what could *possibly* happen on other components. Given an observation on a local component, a decentralized monitor computes the *set of states that are possible* with this observation, and refines (i.e., eliminate possible states) when communicating with other monitors (as we shall see in Sec. 5).

Given a centralized monitor, we define its decentralized version as follows.

Definition 3 (Decentralized Monitor). *Given a centralized monitor* $(Q, \Sigma, q_0, \delta, \mathrm{verdict})$*, the associated decentralized monitor is a 5-tuple* $(2^Q \setminus \{\emptyset\}, (2^{[1;n]} \setminus \{\emptyset\}) \times \Sigma, \{q_0\}, \Delta_\delta, \mathrm{verdict}_D)$ *where:*

- $(2^{[1;n]} \setminus \{\emptyset\}) \times \Sigma$ *is the alphabet,*
- $\Delta_\delta : (2^Q \setminus \{\emptyset\}) \times (2^{[1;n]} \setminus \{\emptyset\}) \times \Sigma \to (2^Q \setminus \{\emptyset\})$ *is the decentralized transition function defined as:*
  $\Delta_\delta(\mathcal{Q}, s, \sigma) = \{q' \in Q \mid \exists \sigma' \in \Sigma, \exists q \in \mathcal{Q} : \sigma = \sigma' \cap \bigcup_{j\in s} AP_j \wedge q' = \delta(q, \sigma')\}$,
- $\mathrm{verdict}_D : (2^Q \setminus \{\emptyset\}) \to \mathbb{B}$ *is the decentralized verdict function, s.t.:*

$$\mathrm{verdict}_D(\mathcal{Q}) = \begin{cases} b \text{ if } \exists b \in \mathbb{B} : \{\mathrm{verdict}(q) \mid q \in \mathcal{Q}\} = \{b\}, \\ ? \text{ otherwise}, \end{cases}$$

for any $\mathcal{Q} \in 2^Q \setminus \{\emptyset\}$.

Intuitively, a decentralized monitor "estimates" the global state that would be obtained by a centralized monitor observing the events produced on all components. The estimation of the global state is modeled by a set of possible states (of the centralized monitor) given the (local) information received so far. When a decentralized monitor receives an event (s, σ), it is informed that the union of the atomic propositions that occurred on the components indexed in the set s is σ. The transition function is s.t. if the estimated global state is $\mathcal{Q} \in 2^Q \setminus \{\emptyset\}$ and it receives (s, σ) as event, then the estimated global state changes to $\Delta_\delta(\mathcal{Q}, s, \sigma)$ which contains all states s.t. one can find a transition in δ from a state in $\mathcal{Q}$ labeled with a global event σ' compatible with σ. In other words, if the actual global state belongs to $\mathcal{Q}$, and the union of events that happen on components indexed in s is σ, then the actual global state belongs to $\Delta_\delta(\mathcal{Q}, s, \sigma)$ which is the set of states that can be reached from a state in $\mathcal{Q}$ with all possible global events (obtained by any observation that could happen on components indexed in $[1; n] \setminus s$). Regarding verdicts, a decentralized monitor emits the same verdict as a centralized one when the current state contains states of the centralized monitor that evaluate on the same verdict.

Example 2 (Decentralized Monitor). Let us consider again the architecture and language L_1 of Example 1. Consider what happens initially on any of the components executing DM_1, the decentralized version of CM_1, see Fig. 2. Initially, the estimated global state is $\{q_0\}$. Suppose the monitor is informed that $\{a\}$ occurred on component

C_A (of index 1), then it will change its estimated global state to $\Delta_{\delta_1}(\{q_0\},\{1\},\{a\}) = \{q_0, q_1\}$. Intuitively, this transition can be understood as follows. Knowing that $\{a\}$ occurred on C_A, the other possible global events are $\{a,b\}$, $\{a,c\}$, and $\{a,b,c\}$, as the monitor does not have information on what happened on C_B and C_C. In CM_1, from state q_0 and these events, states q_0 and q_1 can be reached. Note, the only way to reach $\{q_1\}$ in DM_1, i.e., to know that the global state is q_1 (and is unique), DM_1 has to know that the union of events that occurred on components indexed in $\{1,2,3\}$ is $\{a,b,c\}$.

As illustrated by the example, a decentralized monitor does not depend on the component on which it executes. Its transitions can occur on any component, as it receives an event together with the identifier of components on which such an event occurred. However, a decentralized monitor shall communicate with other decentralized monitors.

5 Communication and Decision Making

Our aim is now to define how a collection of decentralized monitors, analyzing a given distributed trace, should communicate with each other to obtain a verdict in a decentralized manner. The verdict indicates whether the trace, when interpreted as a global trace, is a good or a bad prefix of the language.

5.1 Preliminaries: Local Memory, Clocks, and Communication

Monitor local memory. The local memory of a monitor is a partial function $\mathrm{mem} : \mathbb{N} \to \Sigma \times (2^{[1;n]} \setminus \{\emptyset\})$, purposed to record the "local knowledge" w.r.t. (past instants of) the global (actual) trace produced by the system. If $\mathrm{mem}(t) = (\sigma_t, s_t)$, it means that the monitor knows that the set of all atomic propositions received by the components in s_t is σ_t. Moreover, if $\sigma \in \Sigma$ is the global event at time t and $\mathrm{mem}(t) = (\sigma_t, s_t)$, then $\sigma \cap (\bigcup_{i \in s_t} AP_i) = \sigma_t$. In next section, we will see how after communicating, local monitors can discard elements from their memory.

As a local monitor memorizes the observed local events, it may inform other monitors of the content of its memory via messages. When a monitor receives a memory chunk from another monitor, it merges it with its local memory. For this purpose, for two memories mem and mem', we define the merged memory $\mathrm{mem} \sqcup \mathrm{mem}'$:

$$(\mathrm{mem} \sqcup \mathrm{mem}')(t) = \begin{cases} \mathrm{mem}(t) \cup \mathrm{mem}'(t) & \text{if } t \in \mathrm{dom}(\mathrm{mem}) \cap \mathrm{dom}(\mathrm{mem}'), \\ \mathrm{mem}'(t) & \text{if } t \in \mathrm{dom}(\mathrm{mem}') \setminus \mathrm{dom}(\mathrm{mem}), \\ \mathrm{mem}(t) & \text{otherwise,} \end{cases}$$

where the union $(\sigma, s) \cup (\sigma', s')$ between two memory elements (σ, s) and (σ', s') is defined as $(\sigma \cup \sigma', s \cup s')$. For instance, consider $\mathrm{mem} = \{0 \mapsto (\{b\},\{1,2\}), 1 \mapsto (\{a,b\},\{1,2\}), 2 \mapsto (\emptyset,\{2\})\}$ and $\mathrm{mem}' = \{1 \mapsto (\{c\},\{3\}), 2 \mapsto (\{c\},\{3\})\}$, we have $\mathrm{mem} \sqcup \mathrm{mem}' = \{0 \mapsto (\{b\},\{1,2\}), 1 \mapsto (\{a,b,c\},\{1,2,3\}), 2 \mapsto (\{c\},\{2,3\})\}$.

Monitor local clocks. Each local monitor carries two local (discrete) clocks t and t_{last}. The purpose of t is simply to store the time instant of the last received event from the local component. The purpose of t_{last} is to store the time instant for which it knows the global state of the system. Indeed, the decentralized monitoring algorithm presented in next section will ensure that, on each monitor M_i, for a global trace τ:

- the last event σ emitted by the local component was at time $t : \sigma = \tau(t)$.
- the current state is the state corresponding to $t_{\mathrm{last}} : q = \delta(q_0, \tau[0; t_{\mathrm{last}} - 1])$;

How monitors communicate. As mentioned before, local monitors are required to communicate with each other to share collected information (from their local observation or other monitors). To ensure that communication between monitors aggregates correctly information over time, we suppose having two functions $\mathrm{leader\_mon}$ and $\mathrm{choose\_mon}$ that can be defined e.g., according to the architecture and possibly changing over time.

The function $\mathrm{choose\_mon} : [1;n] \to [1;n]$ indicates for each monitor, the monitor it should communicate with. Local monitors are referred to by their indexes. For information to aggregate correctly, we require $\mathrm{choose\_mon}$ to be bijective, and such that $\forall i \in [1;n], \forall k \in [1;n-1] : \mathrm{choose\_mon}^k(i) \neq i$ where $\mathrm{choose\_mon}^k(i) = \underbrace{\mathrm{choose\_mon}(\ldots(\mathrm{choose\_mon}(i))\ldots)}_{k \text{ times}}$. One can consider for instance $\mathrm{choose\_mon}(i) = (i \mod n)+1$. Note: these requirements are not limitations of our framework but rather guidelines for configuring the communication of our monitors where the architecture is such that a bidirectional direct communication exists between any two components. The proposed algorithms can be easily adapted to any other architecture, provided that a bidirectional communication path exists between any two components (which otherwise would limit the interest of decentralized monitoring).

The function $\mathrm{leader\_mon} : [1;n] \to \{true, false\}$ indicates whether the monitor on the component of the given index is a leader. When receiving new events from the system, only leader monitors can send the local events received from their components. The number of leader monitors influences communication metrics of the monitoring algorithm (see Sec. 7). Using a function makes the algorithm generic and allows leader monitors to change over time.

5.2 Decentralized Monitoring Algorithm

Let us now present the main algorithm for decentralized monitoring. The algorithm is executed independently on each component until there is no event to read and the local monitor has determined the global state, which is given by the condition $t_{\mathrm{last}} > t$ (the time instant corresponding to the last known global state is greater than the time instant of the last received event from the local component).

At an abstract level, the algorithm is an execution engine using a decentralized monitor as per Definition 3. It computes the locally estimated global state of the system by aggregating information from events read locally and partial traces received from other monitors. It stores in q the last known global state of the system at time t_{last}, and in t the time instant of the last event received from the system. The main steps of the algorithm can be summarized as follows:

Algorithm DM (*Decentralized Monitoring*). Let L be the monitored language and q_0 the initial state of its associated centralized monitor. Initialize variables q to q_0, t_{last} to 0, and t to -1. Then, repeat the following steps until the end of the trace and $t_{\mathrm{last}} > t$.

DM1 [Wait] for something from the outside: either an event σ from the system or a message from another monitor (a pair $(q', t_{\mathrm{new}}) \in Q \times \mathbb{N}$ or a partial memory m).

DM2 [Update] If an event (resp. a trace) is received from a component (resp. another monitor), update memory and t. If a state is received, update the known global state.

DM3 [Compute new state] Using the transition function of the decentralized monitor (Definition 3) and the local memory between t_{last} and t, compute the set of possible states. If the set of possible states is a singleton, q and t_{last} are updated.

DM4 [Evaluate and return] If a definitive verdict ($\top$ or $\bot$) is found, return it (and inform other monitors).

DM5 [Prepare communication] Prepare a message to be sent. If a state is received or a new state has been computed (i.e., if q and t_{last} have been modified), append it to the message together with t_{last}. If there are events that occurred after the last found state ($t \geq t_{\text{last}}$), append them to the message, provided that the monitor is a leader ($\text{leader\_mon}(i) = \mathit{true}$) or these events come from another monitor.

DM6 [Communicate] If there is a non-empty message to be sent, then send it to the associated monitor (as determined by function $\text{choose\_mon}(i)$).

At a concrete level, the abstract algorithm is realized in Algorithms 1, 2, and 3. These algorithms execute in the same memory space, and variables are global. The receive function (Algorithm 1) realizes steps **DM1** and **DM2** where i) events and messages from other monitors are received, and, ii) the memory and current state are updated. The receive function is called by the main loop (Algorithm 3) and blocks the execution until an input is received. It can receive three possible inputs (and any combination of them): an event σ from the component (then it updates mem and t), a state q' from another monitor (then it updates q and t_{last} if it does not have fresher information), a partial memory m from another monitor (then it updates mem), or both a state and a partial memory. The function also keeps track of whether a state or a partial memory was received using two Booleans rcv_state and rcv_mem. The update_state function (Algorithm 2) realizes step **DM3** by implementing the transition function Δ_δ of the decentralized monitor using at the same time the local memory mem for efficiency reasons. Variable q keeps track of the last know global state (at time t_{last}. Variable $\mathcal{Q}$ is a temporary variable that keeps track of the set of possible states. Variable upd_state is set to true if the execution of update_state function allows to update the last know global state. The main loop (Algorithm 3) realizes steps **DM4**, **DM5**, and **DM6** where the message is built. Step **DM4** is realized by lines 8 to 11, where, if a new global state is known (either computed with update_state or received in a message), then it is checked if the associated verdict is definitive. The new state together with t_{last} are added to the message. Then, when there are some local events to be shared ($t_{\text{last}} \leq t$), if the monitor received a partial memory or the monitor is a leader (line 12), the partial memory from t_{last} to t (i.e., $\text{mem}(t_{\text{last}}, t)$) and the value of t_{last} are added to the message (line 13). Finally (lines 14-15), the (non-empty) message is sent to the monitor of index $\text{choose\_mon}(i)$.

Remark 2 (Domain of mem*).* At any moment, the only used elements of mem are those between t_{last} and t. Thus, after each step of the algorithm, elements before t_{last} can be discarded. Thus, $\text{dom}(\text{mem}) = [t_{\text{last}}; t]$ is of bounded size under certain conditions discussed in Sec. 6.

The following example illustrates the decentralized monitoring algorithm. Local monitors keep in memory only the events occurring at time instants within $[t_{\text{last}}; t]$.

```
1  (rcv_mem, rcv_state) ⟵ (false, false)
2  when an event σ ∈ Σ_i is received from component:
3      t ⟵ t + 1
4      mem ⟵ mem ⊔ [t ↦ (σ, {i})]
5  when a state q' ∈ Q is received with time t_new:
6      if t_new > t_last then
7          (q, t_last) ⟵ (q', t_new)
8          rcv_state ⟵ true
9  when a partial memory m ∈ ℕ → Σ × (2^[1;n] \ {∅}) is received:
10     mem ⟵ mem ⊔ m
11     rcv_mem ⟵ true
```

Algorithm 1: function receive

```
1  Q ⟵ {q}
2  upd_state ⟵ false
3  for t' from t_last to t do
4      (σ, s) ⟵ mem(t')
5      Q ⟵ Δ_δ(Q, s, σ)
6      if ∃q' ∈ Q : Q = {q'} then
7          (q, t_last) ⟵ (q', t' + 1)
8          upd_state ⟵ true
```

Algorithm 2: function update_state

```
1  (t_last, t) ⟵ (0, −1)
2  (q, Q) ⟵ (q_0, {q_0})
3  mem ⟵ { }
4  repeat until the end of the trace and t_last > t:
5      initialize message
6      receive()
7      update_state()
8      if upd_state ∨ rcv_state then
9          if verdict(q) ∈ {⊤, ⊥} then
10             return verdict(q)
11         add (q, t_last) to message
12     if t_last ≤ t ∧ (rcv_mem ∨ leader_mon(i)) then
13         add (mem(t_last, t), t_last) to message
14     if message is not empty then
15         send message to M_choose_mon(i)
16 return verdict(q)
```

Algorithm 3: Decentralized monitoring algorithm executing on C_i (main loop)

Example 3 (Decentralized Monitoring). Let us go back to the monitoring of the specification introduced in Example 1 and see how this specification is monitored with Algorithms 1, 2, and 3. Table 1 shows how the situation evolves on all three monitors when monitoring the global trace $\emptyset \cdot \{a, b\} \cdot \{a, b, c\} \cdot \{a\}$. As mentioned earlier, the sequence of states of the centralized monitor is $q_0 \cdot q_0 \cdot q_1 \cdot q_1$, and the verdict associated to this trace is $\top$, obtained after the third event. For this example, $\text{leader\_mon}(i) = (i = 1)$ and $\text{choose\_mon}(i) = (i \mod 3) + 1$. For simplicity, in this example, communication between monitors and events from the system occur at the same rate. Cells are colored

in grey when a communication occurs between monitors or an event is read from a component. On each monitor, between any two communications or event receptions, the local memory is represented on two lines: first the values of t_{last}, t, and q the last determined global state, and second the memory content.

- Initially, on each monitor, $t = -1$ (no event received), $t_{\text{last}} = 0$ (the time instant of the last known state), the last know global state is q_0, and the memory is empty.
- When the global event $\emptyset$ occurs, each monitor $M_i, i \in [1;3]$ receives the corresponding local event and records in its memory: $\{0 \mapsto (\emptyset, \{i\})\}$. According to `update_state`, all monitors are able to determine that the global state is (still) q_0, and they update t to 1 and t_{last} to 1 and discards the information about the local received event in memory. Then, each monitor M_i sends the information about its computed state to monitor $M_{\text{choose\_mon}(i)}$. Upon the reception of their message, there is no change in the state of local monitors: the values of t and t_{last} remain the same, and the memory remains empty (the information about the event received at $t = 0$ was discarded because the monitors were able to compute the global state at $t = 0$).
- The remaining steps execute similarly until all monitors return $\text{verdict}(q_1) = \top$.

Remark 3 (Optimizations). Further optimizations can be taken into account in the algorithm. For instance, using a history of sent messages, monitors can remove information from some messages addressed to another monitor, if they already sent this information in a previous message. Further studies are needed to explore the trade-off between local memory consumption vs the size of exchanged messages in the system.

6 Semantics and Properties of Decentralized Monitoring

In this section, we discuss further the semantics induced by the decentralized monitoring algorithm and its properties.

Definition 4 (Semantics of Decentralized Monitoring). *Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the set of system components, $L \subseteq (2^{AP})^*$ be a regular language, and $\mathcal{M} = \{M_1, \ldots, M_n\}$ be the set of component monitors. Further, let $\tau = \tau_1(0) \cup \ldots \cup \tau_n(0) \cdot \tau_1(1) \cup \ldots \cup \tau_n(1) \cdots \tau_1(t) \cup \ldots \cup \tau_n(t)$ be the global behavioral trace, at time $t \in \mathbb{N}$. If some component C_i, with $i \leq n$, M_i has a local state $\mathcal{Q}$ s.t. $\text{verdict}_D(\mathcal{Q}) = \top$ (resp. $\bot$), then $\tau \models_D L = \top$ (resp. $\bot$). Otherwise, $\tau \models_D L =?$.*

By $\models_D$ we denote the satisfaction relation on finite traces in the decentralized setting to differentiate it from the centralized one. Obviously, $\models_C$ and $\models_D$ both yield values from the same truth-domain. However, the semantics are not equivalent, since the current state of the decentralized monitor can contain several states of the centralized one, when a local component has not enough information to determine a verdict. This feature was illustrated in Example 3 where at $t = 2$, the global trace is $\emptyset \cdot \{a, b\} \cdot \{a, b, c\}$, which is a good prefix of the monitored language, only reported at $t = 4$ by Monitor 2.

The precise relation between the centralized and decentralized semantics is given by the two following theorems.

Theorem 1 (Soundness). *Let $L \subseteq \Sigma^*$ and $\tau \in \Sigma^*$, then $\tau \models_D L = \top/\bot \Rightarrow \tau \models_C L = \top/\bot$, and $\tau \models_C L = ? \Rightarrow \tau \models_D L = ?$.*

Soundness states that i) all definitive verdicts found by the decentralized monitoring algorithm are actual verdicts that would be found by a centralized monitor, having access

Table 1: Decentralized monitoring of L_1 on 3 components

Monitor 1			**Monitor 2**			**Monitor 3**		
$t_{\text{last}} = 0$	$t = -1$	$q = q_0$	$t_{\text{last}} = 0$	$t = -1$	$q = q_0$	$t_{\text{last}} = 0$	$t = -1$	$q = q_0$
$\{\ \}$			$\{\ \}$			$\{\ \}$		
Read event $\emptyset$			Read event $\emptyset$			Read event $\emptyset$		
$t_{\text{last}} = 1$	$t = 0$	$q = q_0$	$t_{\text{last}} = 1$	$t = 0$	$q = q_0$	$t_{\text{last}} = 1$	$t = 0$	$q = q_0$
$\emptyset$			$\emptyset$			$\emptyset$		
Send to M_2 $(q_0, 1)$			Send to M_3 $(q_0, 1)$			Send to M_1 $(q_0, 1)$		
$t_{\text{last}} = 1$	$t = 0$	$q = q_0$	$t_{\text{last}} = 1$	$t = 0$	$q = q_0$	$t_{\text{last}} = 1$	$t = 0$	$q = q_0$
$\emptyset$			$\emptyset$			$\emptyset$		
Read event $\{a\}$			Read event $\{b\}$			Read event $\emptyset$		
$t_{\text{last}} = 1$	$t = 1$	$q = q_0$	$t_{\text{last}} = 1$	$t = 1$	$q = q_0$	$t_{\text{last}} = 2$	$t = 1$	$q = q_0$
$\{1 \mapsto (\{a\}, \{1\})\}$			$\{1 \mapsto (\{b\}, \{2\})\}$			$\emptyset$		
Send to M_2 $((\{a\}, \{1\}), 1)$			Send to M_3 $((\{b\}, \{2\}), 1)$			Send to M_1 $(q_0, 2)$		
$t_{\text{last}} = 2$	$t = 1$	$q = q_0$	$t_{\text{last}} = 1$	$t = 1$	$q = q_0$	$t_{\text{last}} = 2$	$t = 1$	$q = q_0$
$\emptyset$			$\{1 \mapsto (\{a, b\}, \{1, 2\})\}$			$\emptyset$		
Read event $\{a\}$			Read event $\{b\}$			Read event $\{c\}$		
$t_{\text{last}} = 2$	$t = 2$	$q = q_0$	$t_{\text{last}} = 1$	$t = 2$	$q = q_0$	$t_{\text{last}} = 2$	$t = 2$	$q = q_0$
$\{2 \mapsto (\{a\}, \{1\})\}$			$\{1 \mapsto (\{a, b\}, \{1, 2\}),$ $2 \mapsto (\{b\}, \{2\})\}$			$\{2 \mapsto (\{c\}, \{3\})\}$		
Send to M_2 $(q_0, 2), ((\{a\}, \{1\}), 2)$			Send to M_3 $((\{a, b\}, \{1, 2\}), (\{b\}, \{2\}), 1)$			Send to M_1 $((\{c\}, \{3\}), 2)$		
$t_{\text{last}} = 2$	$t = 2$	$q = q_0$	$t_{\text{last}} = 2$	$t = 2$	$q = q_0$	$t_{\text{last}} = 2$	$t = 2$	$q = q_0$
$\{2 \mapsto (\{a, c\}, \{1, 3\})\}$			$\{2 \mapsto (\{a, b\}, \{1, 2\})\}$			$\{2 \mapsto (\{b, c\}, \{2, 3\})\}$		
Read event $\{a\}$			Read event $\emptyset$			Read event $\emptyset$		
$t_{\text{last}} = 2$	$t = 3$	$q = q_0$	$t_{\text{last}} = 2$	$t = 3$	$q = q_0$	$t_{\text{last}} = 2$	$t = 3$	$q = q_0$
$\{2 \mapsto (\{a, c\}, \{1, 3\}),$ $3 \mapsto (\{a\}, \{1\})\}$			$\{2 \mapsto (\{a, b\}, \{1, 2\}),$ $3 \mapsto (\emptyset, \{2\})\}$			$\{2 \mapsto (\{b, c\}, \{2, 3\}),$ $3 \mapsto (\emptyset, \{3\})\}$		
Send to M_2 $((\{a, c\}, \{1, 3\}), (\{a\}, \{1\}), 2)$			Send to M_3 $(q_0, 2),$ $((\{a, b\}, \{1, 2\}), (\emptyset, \{2\}), 2)$			Send to M_1 $((\{b, c\}, \{2, 3\}), (\emptyset, \{3\}), 2)$		
$t_{\text{last}} = 4$	$t = 3$	$q = q_0$	$t_{\text{last}} = 4$	$t = 3$	$q = q_1$	$t_{\text{last}} = 4$	$t = 3$	$q = q_0$
$\emptyset$			$\emptyset$			$\emptyset$		
Return $\text{verdict}(q_1) = \top$			Return $\text{verdict}(q_1) = \top$			Return $\text{verdict}(q_1) = \top$		

to the global trace, and ii) decentralized monitors do not find more definitive verdicts ($\top$ or $\bot$) than the centralized one.

Theorem 2 (Completeness). *Let $L \subseteq \Sigma^*$ and $\tau \in \Sigma^*$, then $\tau \models_C L = \top/\bot \Rightarrow \exists \tau' \in \Sigma^* : \tau \cdot \tau' \models_D L = \top/\bot$.*

Completeness states that all verdicts found by the centralized algorithm for some global trace τ will be eventually found by the decentralized algorithm on a continuation $\tau \cdot \tau'$. Generally, when the rate of communication between monitors (compared to the reception of events) is unknown or when not all monitors are leaders, it is not possible to determine the maximal length of τ'. When monitors communicate at the same rate as monitors receive events and all monitors are leaders (i.e., they can send message spontaneously – $\text{leader\_mon}(i) = \mathit{true}$, for any $i \in [1; n]$), then, as was the case

in [10], we can bound the maximal length of τ' by n (the number of components in the system), which also represents the maximal delay, induced by decentralized monitoring.

Theorem 3 (Completeness with bounded delay). *Let $L \subseteq \Sigma^*$ and $\tau \in \Sigma^*$, if monitors receive events and communicate at the same rate and if all monitors are leaders, then $\tau \models_C L = \top/\bot \Rightarrow \exists \tau' \in \Sigma^* : |\tau'| \leq n \wedge \tau \cdot \tau' \models_D L = \top/\bot$.*

7 Implementation and Experimental Results

We present DECENTMON2 a new benchmark tool used to evaluate decentralized monitoring (Sec. 7.1) using specifications given as LTL formulas (Sec. 7.2) and specifications patterns (Sec. 7.3). Then, we draw conclusions from our experiments (Sec. 7.4). Further experimental results are available at [17].

7.1 DECENTMON2: a Benchmark for Generalized Decentralized Monitoring

DECENTMON2 is an benchmark dedicated to decentralized monitoring. DECENTMON2 consists of: a completely redeveloped version of DECENTMON [10], an implementation of the decentralized monitoring algorithm presented in Sec. 5.2, a trace generator, and an LTL-formula generator. DECENTMON2 consists of 1,300 LLOC, written in the functional programming language OCaml. It can be freely downloaded and run from [17].

The system takes as input multiple traces (that can be automatically generated), corresponding to the behavior of a distributed system, and a specification given by a deterministic finite-state automaton. Then the specification is monitored against the traces in two different modes: a) by merging the traces to a single, global trace and then using a "centralized monitor" for the specification (i.e., all components send their respective events to the central monitor who makes the decisions regarding the trace), b) by using the decentralized version introduced in [10], and c) by using the decentralized approach introduced in this paper (i.e., each trace is read by a local monitor in the two last cases). To favor the centralized case, monitors send their events only if they differ from the previous one, which decreases the number of exchanged messages. We have evaluated the three different monitoring approaches (i.e., centralized vs. LTL-decentralized vs generalized-decentralized) using several set-ups described in the remainder of this section. To compare monitoring metrics obtained with the decentralized algorithm in [10] and the one in this paper, we used LTL2MON [18], to convert LTL formulas into automata-based (centralized) monitors. For our comparison purposes, we used results on common LTL formulas and traces using the experimental setup depicted in Fig. 3. For each of the metric mentioned in the following sections, ratios are obtained by dividing the value obtained in the decentralized case over the value obtained in the centralized case.

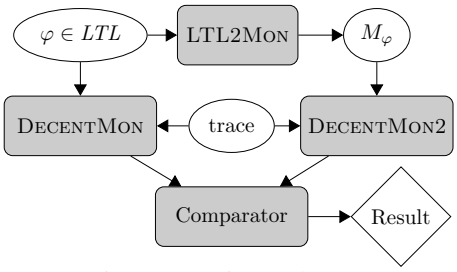

Fig. 3: Experimental setup

To compare with the decentralized monitoring algorithm obtained in [10], the emission of events occurs at the same rate as the communication between monitors. Recall that it was assumed in [10] whereas our monitoring algorithm allows different ratios.

Each line of the following arrays is obtained by conducting 1,000 tests, each with a fresh trace of 1,000 events and specification. We use the same architecture as in the running example. Note that benchmarks with different architectures and rates of communication/event-emission were also conducted, and are available from [17].

For the following monitoring metrics, we measure the size of the elements exchanged by monitors as follows. Suppose we monitor an LTL formula φ over AP with an automaton defined over the alphabet $\Sigma = 2^{AP}$ with set of states Q: each event is of size $\lceil \log_2 |\Sigma| \rceil$, each state is of size $\lceil \log_2 |Q| \rceil$, each time unit t is of size $\lceil \log_2(t) \rceil$, each formula is of size $n \times \lceil \log_2(|AP| + |Op|) \rceil$ where n is the number of symbols in the formula, AP is the set of atomic propositions of the formula and $Op = \{\top, \bot, \vee, \wedge, \neg, \Rightarrow, \Leftrightarrow, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}, \mathbf{W}, \overline{\mathbf{X}}, \#, (,)\}$ is the set of symbols in formulas handled by DECENTMON. Then in the following tables, the following metrics are used: $\#\mathrm{msg.}$, the total number of exchanged messages, $|\mathrm{msg.}|$, the total size of exchanged messages (in bits), $|\mathrm{trace}|$ the size of the prefix of the trace needed to obtain a verdict, delay, the number of additional events needed by the decentralized algorithm to reach a verdict compared to the centralized one, $|\mathrm{mem}|$, the memory in bits needed for the structures (i.e., formulas for [10], partial function mem plus state for our algorithm).

7.2 Benchmarks for Randomly Generated formulas

For each size of formula (from 1 to 6), DECENTMON2 randomly generated 1,000 formulas in the architecture described in Example 1. How the three monitoring approaches compared on these formulas can be seen in Tables 2 and 3. The first column of these tables shows the size of the monitored LTL formulas. Note, our system measures formula size in terms of operator entailment[1] inside it (state formulas excluded), e.g., $\mathbf{F}a \wedge \mathbf{G}(b \wedge c)$ is of size 2.

Table 2: Number and size of messages - random formulas

$\lvert\varphi\rvert$	$\#\mathrm{msg.}$			$\lvert\mathrm{msg.}\rvert$			$\#\mathrm{msg.}$ ratio		$\lvert\mathrm{msg.}\rvert$ ratio	
	cm	dm1	dm2	cm	dm1	dm2	dm1/cm	dm2/cm	dm1/cm	dm2/cm
1	3.49	1.13	3.73	10.4	87.2	23.8	0.32	1.06	8.31	2.27
2	4.04	1.89	5.4	12.1	316	39.2	0.46	1.33	26.0	3.23
3	9.33	5.34	16.9	27.9	3,220	166	0.57	1.37	115	4.5
4	25.1	12.6	35.9	75.3	8,430	350	0.5	1.27	112	4.16
5	39.7	21.9	71.0	119	36,500	775	0.55	1.33	306	4.86
6	90.9	47.3	116	272	284,000	1,180	0.52	1.23	1,040	4.21

For example, the last line in Table 2 says that we monitored 1,000 randomly generated LTL formulas of size 6. On average, monitors using the centralized algorithm, the decentralized algorithm using LTL formulas, and the decentralized algorithm using automata, exchanged 90.9, 47.3, 116 messages, had messages of size 272 bits, 284,000 bits, 1180 bits, respectively. The last two pairs of columns show the ratios of the previous metrics ob-

Table 3: Trace length, delay, and memory size - random formulas

$\lvert\varphi\rvert$	$\lvert\mathrm{trace}\rvert$			delay		$\lvert\mathrm{mem}\rvert$	
	cm	dm1	dm2	dm1	dm2	dm1	dm2
1	1.33	1.66	2.61	0.32	1.28	44.2	7.93
2	1.67	2.15	3.2	0.48	1.53	156	9.72
3	5.21	5.79	8.8	0.58	1.6	458	10.4
4	15.7	16.4	19.3	0.7	1.66	1,100	11.3
5	25.5	26.4	36.3	0.82	1.79	2630	12.4
6	59.4	60.2	63.2	0.76	1.66	5,830	12.0

[1] Experiments show that operator entailment is more representative of how difficult it is to progress it in a decentralized manner. formulas of size above 6 are not realistic in practice.

Table 4: Number and size of messages - specification patterns

$\|\varphi\|$	#msg.			|msg.|			#msg. ratio		|msg.| ratio	
	cm	dm1	dm2	cm	dm1	dm2	dm1/cm	dm2/cm	dm1/cm	dm2/cm
abs	7.33	4.46	17.9	22	2,050	194	0.6	2.44	93.6	8.85
exis	43.9	19.7	64.2	131	10,200	663	0.45	1.46	77.6	5.03
bexis	65.3	31.6	379	19.6	1,170,000	5,450	0.48	2.17	5,970	10.4
univ	10.3	5.92	30.9	31	2,750	379	0.57	2.98	88.6	12.2
prec	77.6	25.4	68.1	232	8,710	648	0.32	1.29	37.4	4.11
resp	959	425	1,070	2,870	337,000	9,760	0.44	1.12	117	3.39
precc	7.68	4.81	18.9	23.	5,180	218	0.62	2.47	225	9.53
respc	643	381	732	1,920	719,000	6,680	0.59	1.13	372	3.46
consc	490	201	469	1,470	337,000	4,260	0.41	1.13	229	3.43

tained in the decentralized cases over the centralized one. For instance, the last line in Table 2 says that the decentralized algorithm with LTL formulas induced 0.52 times the number of messages of the centralized algorithm, whereas the decentralized algorithm with automata induced 1.23 times the number of messages. Message ratios and metrics in Table 3 read similarly.

7.3 Benchmarks for Patterns of formulas

We also conducted benchmarks with more realistic specifications, obtained from specification patterns [19]. Actual formulas underlying the patterns are available at [20, 17]. We generated formulas as follows. For each pattern, we randomly select one of its associated formulas. Such a formula is "parametrized" by some atomic propositions. To obtain randomly generated formula, using the distributed alphabet, we randomly instantiate atomic propositions.

Table 5: Trace length, delay, and memory size - specification patterns

$\|\varphi\|$	|trace|			delay		|mem|	
	cm	dm1	dm2	dm1	dm2	dm1	dm2
abs	3.89	4.55	5.66	0.66	1.77	496	12.4
exis	28.2	28.9	29.9	0.65	1.68	376	11.7
bexis	42.6	43.1	116	0.581	1.56	28,200	14.4
univ	5.96	6.73	7.76	0.76	1.79	498	13.0
prec	50.8	51.6	35.5	0.81	1.66	663	11.5
resp	638	639	639	0.32	0.7	1,540	8.61
precc	4.11	4.82	5.72	0.7	1.64	1,200	11.6
respc	427	428	428	0.59	1.16	4,650	10.7
consc	325	325	326	0.6	1.35	2,720	10.8

Results are reported in Tables 4 and 5 for each kind of patterns (absence, existence, bounded existence, universal, precedence, response, precedence chain, response chain, constrained chain), we generated again 1,000 formulas, monitored over the same architecture as used in Example 1.

7.4 Conclusions from the Experiments and Discussion

The number and size of exchanged messages when monitoring with the decentralized algorithm using automata are in the same order of magnitude (and most often lower) as when monitoring with the centralized algorithm. Comparing the decentralized monitoring algorithms, the number of messages when using LTL formulas is always lower but the size of messages is much bigger in that case (sometimes by orders of magnitude). Delays are always greater when using automata but they remain in the same order of

magnitude. Please also note that we have conducted benchmarks where our algorithm uses only one leader monitor, which tends to augment the delay (whereas in the algorithm using LTL formulas monitors are not constrained) - see the discussion below. Regarding the size of memory, the algorithm using automata is always more efficient by several orders of magnitude when the size of formulas grows.

Efficiency of Implementation. Another interesting feature of our algorithm is its usability in implementation. We measured the execution time (in seconds) and real memory consumption of the two (reasonably optimized) implementations of benchmarks (in the same programming language), see Table 6 where $|\mathrm{msg.}|$ is in kb and $|\mathrm{mem}|$ in MB.

Table 6: Performance of implementations

	$\#\mathrm{msg.}$	$\lvert\mathrm{msg.}\rvert$	$\lvert\mathrm{mem}\rvert$	time
DECENTMON	367	$21,667$	$157,845$	4.724
DECENTMON2	$3,258$	59	18	0.064

We only report the results when monitoring formulas of type bounded existence, over the same alphabet as before, with a trace of 10,000 events. For other kinds of formulas, the trend is similar. As expected, progression is certainly more costly and thus less appropriate for monitoring. Moreover, the size of messages (and hence the size of formulas) monitors have to handle becomes unmanageable quite rapidly.

Influence of the number of leaders. We also made some experiments (omitted for space reasons) regarding the influence of the number of leader monitors. It turns out that, as the number of leaders augments in the system, the number of messages augments, whereas the delay induced by decentralized monitoring reduces. For instance, by allowing all monitors to communicate spontaneously (i.e., with $\mathrm{leader\_mon}(i) = \mathit{true}$ for any $i \in [1; n]$), we observed that, for several patterns of formulas, i) a shorter average delay and less memory consumption by a factor of 1.5, and ii) the total size of messages was, in average, multiplied by 1.7 while their number was multiplied by 2 (thus the average size of messages decreased).

8 Future Work

Experiments in Sec. 7 indicate that some parameters of our monitoring algorithm such as the frequency of communication, the number of leader monitors, and the communication architecture, influence monitoring metrics. Our experiments allowed to sketch some empiric laws but a deeper understanding of the influence of each of these parameters is certainly needed to optimize decentralized monitoring on specific architectures.

Another line of research is related to security in decentralized monitoring, when for instance monitoring security-related properties, or when the property involves atomic propositions with confidential information. Decentralized monitoring imposes local monitors to communicate, for instance over some network. Exchanged messages contain information about the observation or state of monitors w.r.t. the property of interest. Some confidentiality issues may arise. Thus, an interesting question is to determine how and to what extent monitors could encode their local observation, transmit the encoded information, so that the message is of benefit to the recipient (in terms of gained information), but not to an external observer.

Communication is constrained by the $\mathrm{choose\_mon}$ function to e.g., reflect architectural constraints. We will determine how to optimize the definition of the $\mathrm{choose\_mon}$ function according to the monitored language, the memory content, or the current state of local monitors so as to minimize the size and number of exchanged messages.

Another extension is to augment local monitors with enforcement primitives [21] to correct violations. For this purpose, monitors can use their estimated global state and release events only when the (estimated) states are associated to a "good" verdict.

References

1. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: 9th Int. Conf. on Software Engineering and Formal Methods. Volume 7041 of LNCS., Springer (2011) 204–220
2. Zhou, W., Sokolsky, O., Loo, B.T., Lee, I.: DMaC: Distributed monitoring and checking. In: 9th Work. on Runtime Verification. Volume 5779 of LNCS., Springer (2009) 184–201
3. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Decentralized runtime analysis of multithreaded applications. In: 20th Parallel and Distributed Processing Symp., IEEE (2006)
4. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symp. on Foundations of Computer Science. (1977) 46–57
5. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers. In: 14th Symp. on Formal Methods. Volume 4085 of LNCS., Springer (2006) 557–572
6. Graf, S., Peled, D., Quinton, S.: Monitoring distributed systems using knowledge. In: Joint 13th IFIP WG 6.1 Int. Conf., FMOODS 2011, and 31st IFIP WG 6.1 Int. Conf., FORTE 2011. Volume 6722 of LNCS., Springer (2011) 183–197
7. Wang, Y., Yoo, T.S., Lafortune, S.: New results on decentralized diagnosis of discrete event systems. In: 42nd Ann. Allerton Conf. on Comm., Control, and Computing. (2004)
8. Cassez, F.: The complexity of codiagnosability for discrete event and timed systems. In: 8th Int. Symp. on Automated Technology for Verification and Analysis. Volume 6252 of LNCS., Springer (2010) 82–96
9. Tripakis, S.: Decentralized observation problems. In: 44th IEEE Conf. Decision and Control, IEEE (2005) 6–11
10. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: 18th Int. Symp. on Formal Methods. Volume 7436 of LNCS., Springer (2012) 85–100
11. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence **22** (1998) 5–27
12. Bartocci, E.: Sampling-based decentralized monitoring for networked embedded systems. In: 3rd Int. Work. on Hybrid Autonomous Systems. Volume 124 of EPTCS. (2013) 85–99
13. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: 26th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science. Volume 4337 of LNCS., Springer (2006) 260–272
14. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: 9th Int. Work. on Runtime Verification. LNCS (2009) 40–59
15. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? Software Tools for Technology Transfert **14** (2012) 349–382
16. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20** (2011) 14
17. Cornebize, T., Falcone, Y.: DECENTMON2 (2013) http://decentmon2.forge.imag.fr.
18. Bauer, A.K.: LTL2MON (2009) http://ltl3tools.sourceforge.net.
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Intl. Conf. on Software Engineering (ICSE), ACM (1999) 411–420
20. Alavi, H., Avrunin, G., Corbett, J., Dillon, L., Dwyer, M., Pasareanu, C.: Specification patterns website (2011) http://patterns.projects.cis.ksu.edu/.
21. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design **38** (2011) 223–262